

ZENON™ IP COMMUNICATIONS PLATFORM  
**ZSDK Mobile Core**

Provides information to Platform Owners and Partners

Regarding how to configure the Zenon Platform to their needs

The ZENON logo and ZENON are the trademarks of Requestec Ltd. All other trademarks are owned by their respective companies.

© 2002-2012 REQUESTEC Ltd.

All rights, including translation into other languages, are reserved under the Universal Copyright Convention and other related instruments. Any use of this documentation and/or its associated software, including reproduction, modification, distribution, republication, transmission, re-transmission, or public showing, without the prior written permission of an authorised REQUESTEC representative is prohibited.

This document and any software and programs mentioned in this document, whether delivered electronically or via other media, are the sole property of REQUESTEC Ltd. and are sold under a license agreement. Reproduction or modification of the software or documentation by any means is prohibited without the express written consent of an authorised REQUESTEC representative.

This document has been created by:

Pankaj Sharma ([pankajs@requestec.com](mailto:pankajs@requestec.com))

Table of Contents

[1. Introduction 6](#_Toc379806785)

[2. System Requirements 6](#_Toc379806786)

[3. What’s in the Box 6](#_Toc379806788)

[4. Installing the SDK 7](#_Toc379806790)

[4.1 Loading the Application Package on the device 7](#_Toc379806792)

[4.2 Building the Example Application 7](#_Toc379806793)

[4.3 Using the Zenon SDK in custom application source 8](#_Toc379806794)

[4.4 Initializing the SDK 12](#_Toc379806795)

[4.5 Setting the Central Login Server 12](#_Toc379806796)

[4.6 Login to Zenon Server 13](#_Toc379806797)

[4.7 Make Video/Voice Calls 14](#_Toc379806798)

[4.8 Receive Incoming Video/Voice Calls 15](#_Toc379806799)

[4.9 Accept/Reject Incoming Video/Voice Calls 15](#_Toc379806800)

[4.10 Switch from Speaker to Headphone and back 16](#_Toc379806801)

[4.11 Logout from Zenon SDK 17](#_Toc379806802)

[5 SDK Architecture 18](#_Toc379806803)

[6 Application Requirements 19](#_Toc379806804)

[7 Supported Devices 21](#_Toc379806805)

[8 SDK API Reference 22](#_Toc379806806)

[8.1 Android 22](#_Toc379806807)

[8.1.1 ConfigManager Class 22](#_Toc379806808)

[8.1.2 EventDispatcher Class 22](#_Toc379806809)

[8.1.3 EventsManager Class 23](#_Toc379806810)

[8.1.4 Zebra Class 24](#_Toc379806811)

[8.1.5 ConnectionManager Class 26](#_Toc379806812)

[8.1.6 CallManager Class 27](#_Toc379806813)

[8.1.7 Logger Class 29](#_Toc379806814)

[8.2 iOS 30](#_Toc379806815)

[8.2.1 ConfigManager Class 30](#_Toc379806816)

[8.2.2 EventDispatcher Class 31](#_Toc379806817)

[8.2.3 EventsManager Class 32](#_Toc379806818)

[8.2.4 Zebra Class 32](#_Toc379806819)

[8.2.5 ConnectionManager Class 34](#_Toc379806820)

[8.2.6 CallManager Class 36](#_Toc379806821)

[8.2.7 Logger Class 39](#_Toc379806822)

[Appendix A: Call End Reason Codes 42](#_Toc379806823)

[Appendix B: Zebra API 43](#_Toc379806824)

[Appendix C: Configurable Components (Available upon request) 47](#_Toc379806825)

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Revision** | **Changes** | **Author** |
| 19th December 2013 | 1.0 | First Draft | Pankaj Sharma |
| 04th February 2014 | 2.0 | Modifications for incorporating the configuration parameters in SDK | Pankaj Sharma |

# Introduction

This document provides detailed information on the Zenon Mobile Core SDK for iOS and Android platforms.

The Zenon SDK covers the following features:

* Integration to the Zenon Gateway for Authentication and Zebra APIs (registration, messaging, address book, etc)
* Receiving asynchronous messages from the Zenon Gateway which are fired as events (incoming call, new IM, friend request, etc)
* Voice and video capture, processing and streaming
* Peer 2 Peer and server-routed voice and video calling
* In-call video screen UI
* Dynamic Bandwidth Adaptation for best quality video calls
* Acoustic Echo Cancellation
* File Transfer
* Photo Capture
* Mobile Native Address Book integration
* Video conferencing and multiple phone line control

Some of the features are not documented here but are available on request.

The UI SDK builds on top of the core SDK with models, views and controllers. UI components can be HTML5 or native. For more information on the UI SDK please make a request.

# System Requirements



Android

* A system that runs Eclipse and Android SDK (i.e. Mac, Linux or Windows).
* Latest Android SDK with Android platform 2.1-update1 package.
* Eclipse with latest Android ADT plugin.

iOS

* Intel based Mac
* XCode 4 or later and iOS SDK

# What’s in the Box



Android

The Zenon SDK is shipped as a Zip file containing:

* **ZenonSDKExample** – Eclipse project of worked out example Video/Voice Calling application using ZenonSDK core framework.
* **ZenonSDK.jar** – Jar framework containing ZenonSDK core. This needs to be included in your eclipse project libs.
* **armeabi-v7a/libcommunicatorjni.so** and **armeabi/libcommunicatorjni.so** – Native libraries to be included in your eclipse project libs folder as per the ZenonSDKExample project for linking purposes.
* **Documentation** – Javadoc and this user guide.

iOS

The Zenon SDK is shipped as a Zip file containing:

* **ZenonSDKExample** – XCode project of worked out example Video/Voice Calling application using ZenonSDK core framework.
* **ZenonSDK.a** – Library framework containing ZenonSDK core. This needs to be included in your XCode project.
* **mediaEngine** – Native media management libraries to be included in your XCode project as per the ZenonSDKExample project for linking purposes.
* **Documentation** – Appledoc and this user guide.

# Installing the SDK



## Loading the Application Package on the device

Android

The Zenon SDK for Android is shipped as a zipped up framework. Unpack this and you will see a file named ZenonSDKExample.apk. You can load this to any of your android devices through standard android application loading/installation procedures.

iOS

The Zenon SDK for Android is shipped as a zipped up framework. Unpack this and you will see a file named ZenonSDKExample.ipa. You can load this to any of your iOS devices through standard iOS application loading/installation procedures.

## Building the Example Application

Android

The Zenon SDK for Android contains an eclipse project. Import that project into your eclipse workspace and build it. The Android development setup is ready and you can build and download the apk to your phone using eclipse ADT. For more information on how to build an android application using eclipse, please visit <http://developer.android.com/tools/index.html>

iOS

The Zenon SDK for iOS contains an XCode 5 project named ZenonSDKExample.xcodeproj. Configure the XCode to include:

* A valid provisioning profile.
* Valid Code Signing identity.
* Path to the Keychain containing valid provisioning profile and code signing identity.
* Bundle Identifier.

The ZenonSDKExample directory also contains a build script named build.sh. Modify the names and paths of above four points in this build script as per your development environment.

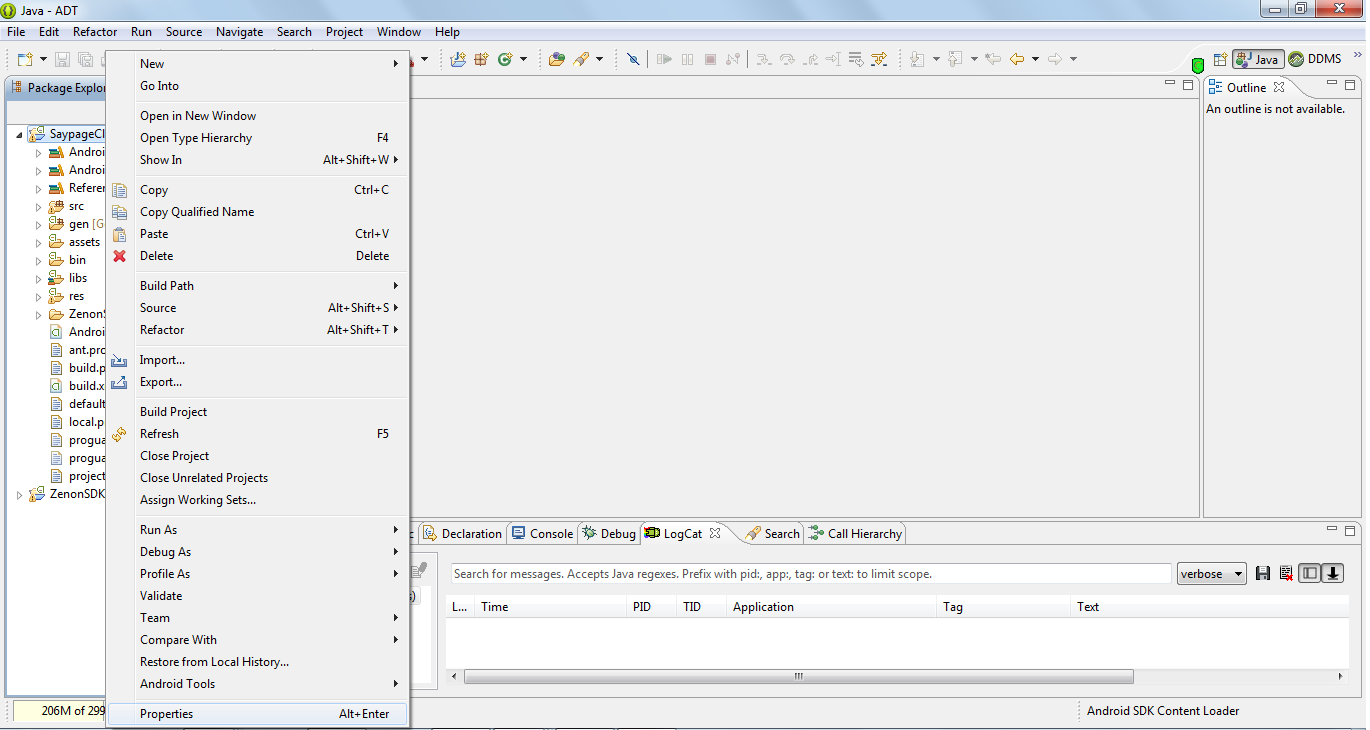
This script can be run from a terminal to build the ZenonSDKExample application.

## Using the Zenon SDK in custom application source

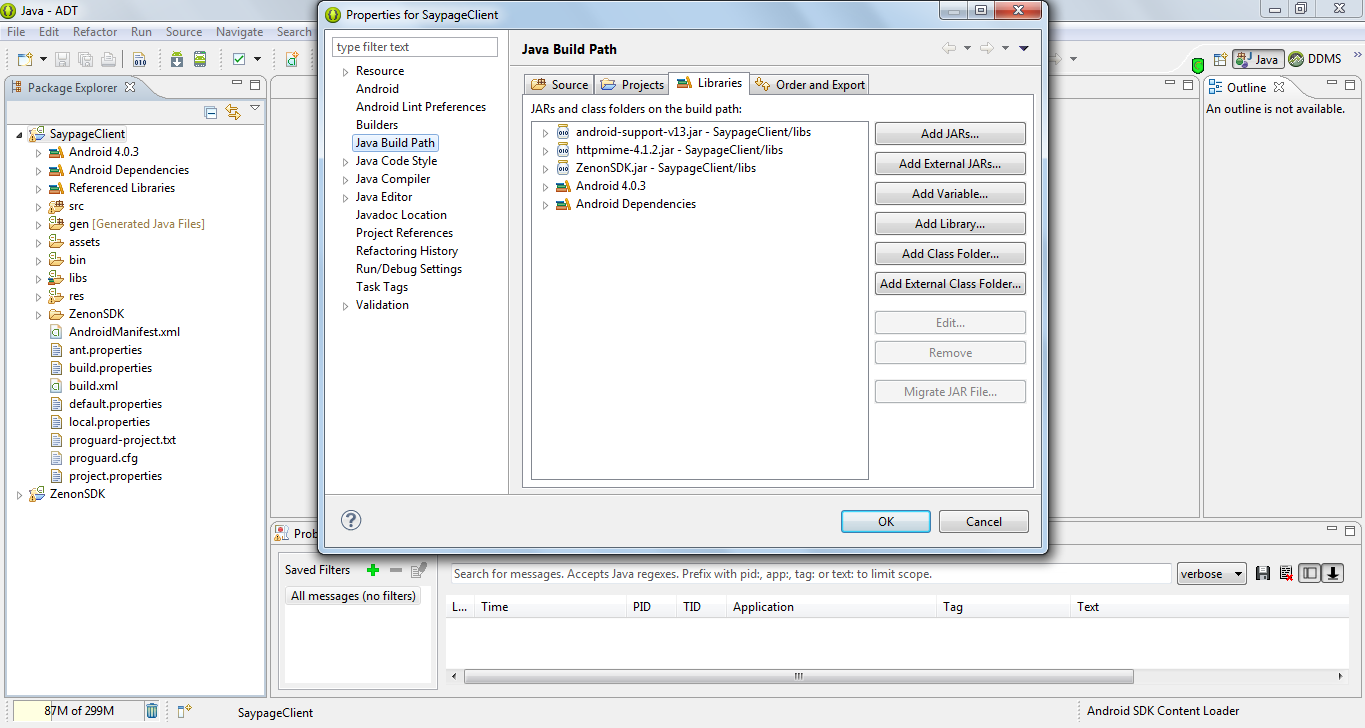
Android

Unpack the supplied Zenon SDK zip wherever you want on your disk and then import the project into your Eclipse. You will have the Zenon Sample Application in your eclipse project and you can run this on your device to test the login/logout and video/voice calling using Zenon SDK.

The unzipped directory contains libs folder in ZenonSDKExample folder. Copy all files from that folder to the libs folder of your application eclipse project. Further, include the ZenonSDK.jar in your class path. The following steps explain what to do in detail:



Step-1 – Right Click your project in eclipse project pane and select Properties from the pop-up list.



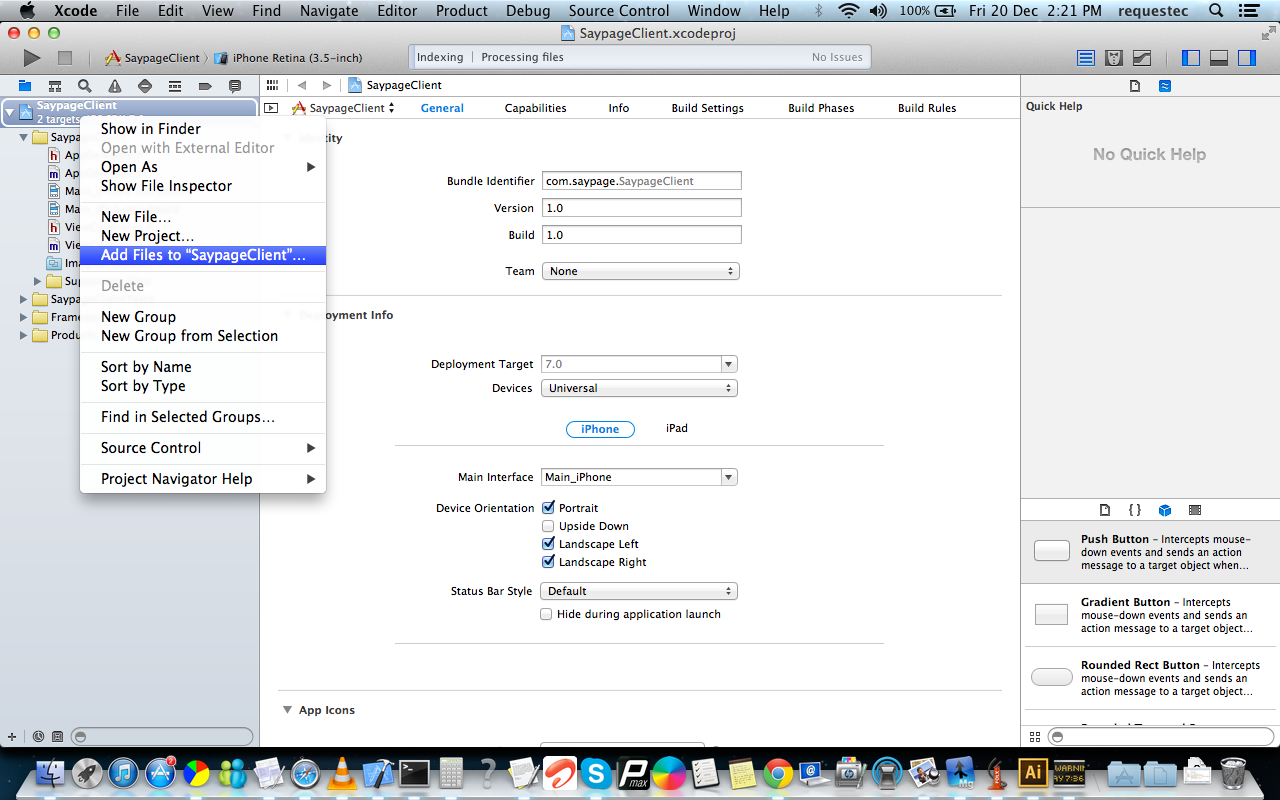
Step-2 – Open project properties and select “Java Build Path”, click “Add JARs...” then select “ZenonSDK.jar” from within the imported ZenonSDKExample project’s “libs” folder and click OK (See Figure above).

Step-3 – Rebuild your project.

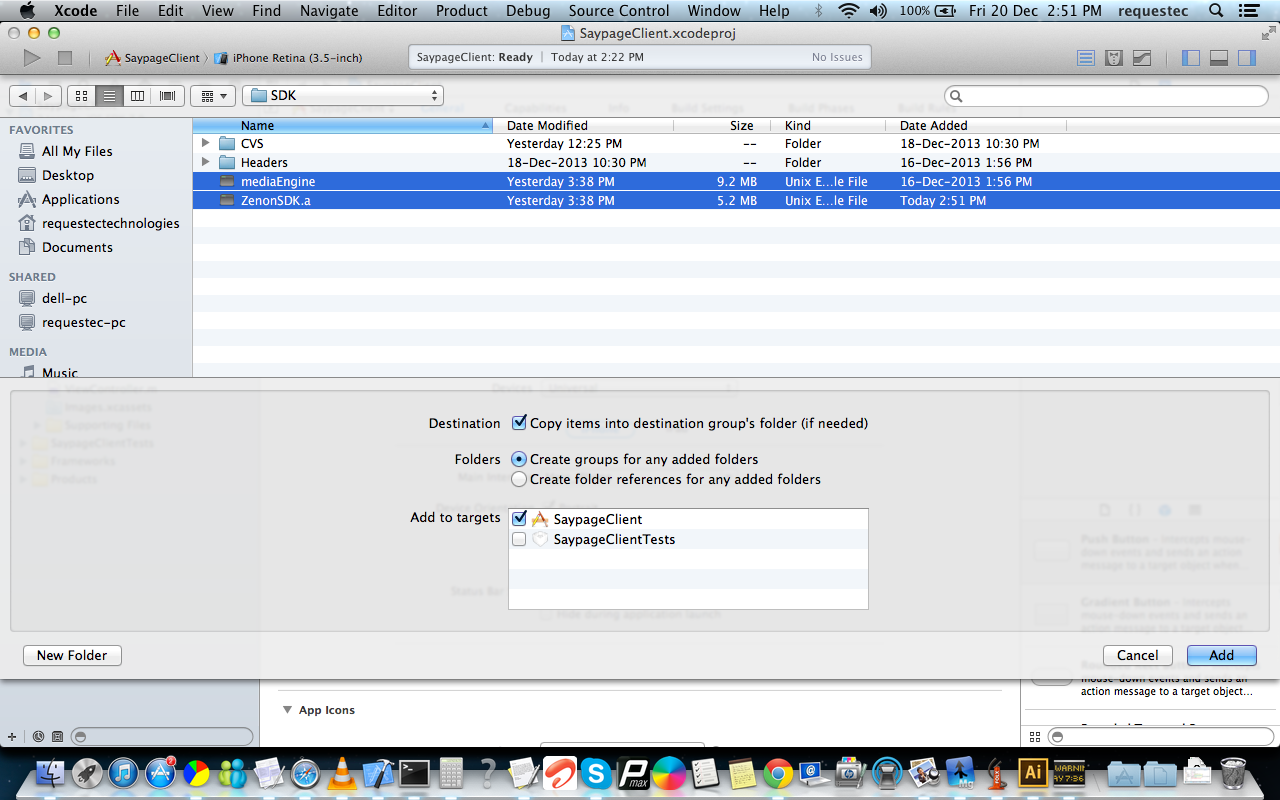
Once these steps have been completed, your project is ready to use the ZenonSDK!

iOS

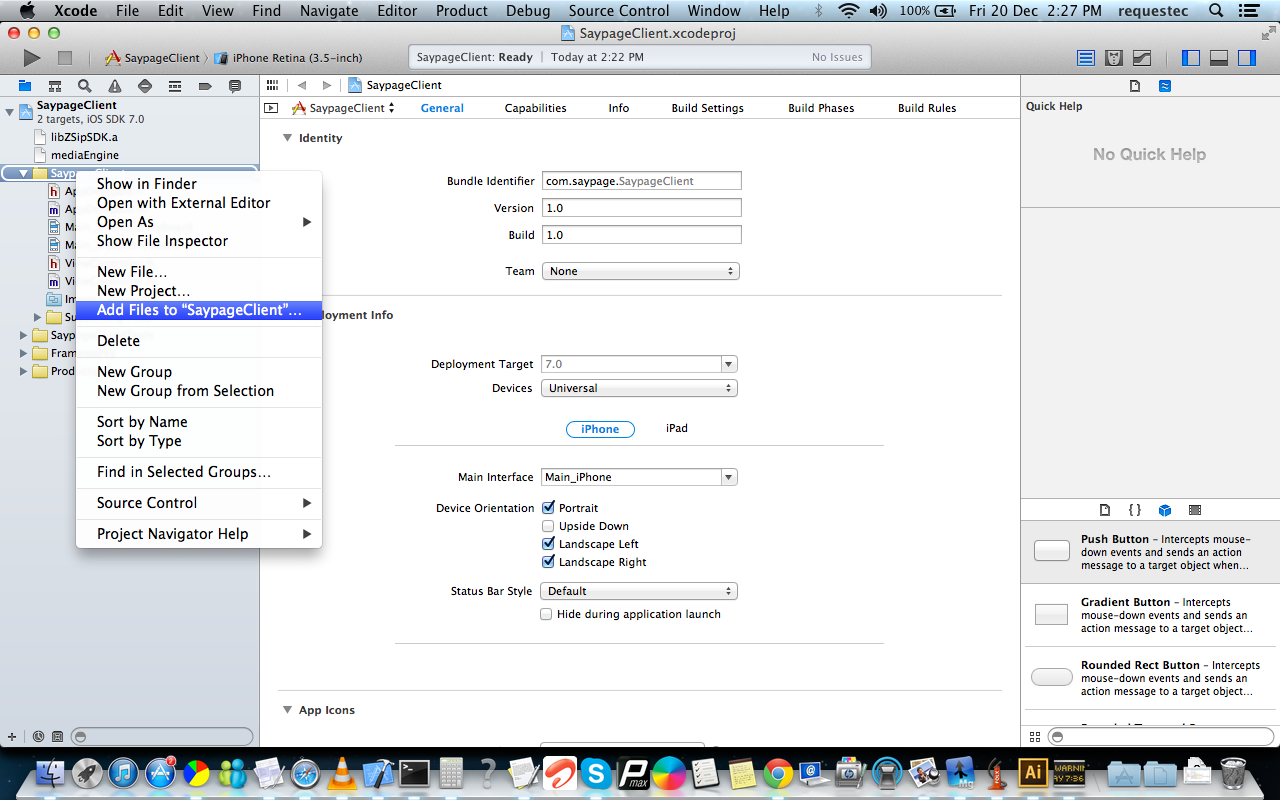
Unpack the supplied Zip file and copy the SDK folder to your project. Add the ZenonSDK.a and mediaEngine files to your application XCode project. Further, add the header files in the Header directory in the SDK folder to your project. The following steps explain what to do in detail:



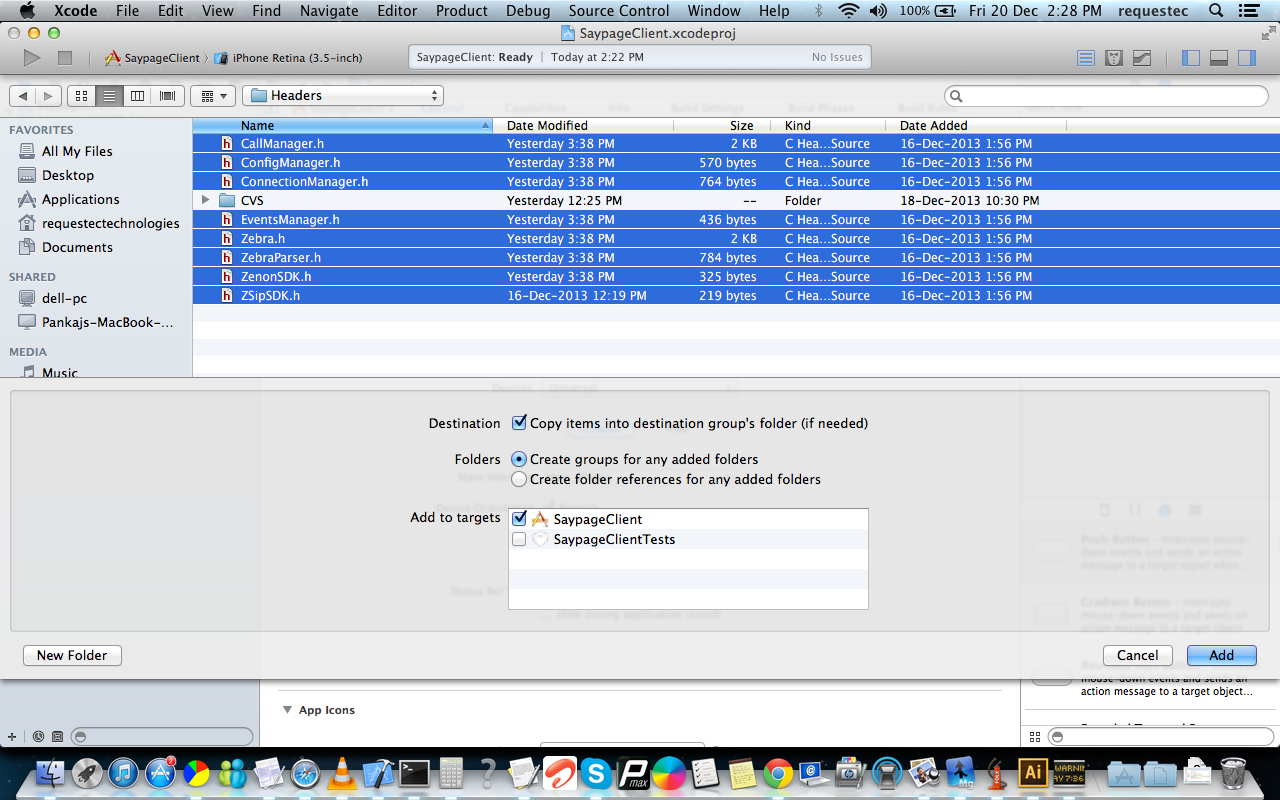
Step-1- Right Click the project in XCode Project pane and select Add Files to “Project Name”.



Step-2 – Browse to the extracted ZenonSDKExample/SDK folder and select ZenonSDK.a and mediaEngine files and click Add.



Step-3- Again right click the project in XCode project pane and select “Add files to Project” option.



Step-4 – Browse to the extracted ZenonSDKExample/SDK/Headers directory and select all header files and click Add. This will add all the required ZenonSDK files to your application project.

Step-5 – Rebuild your project.

Once these steps have been completed, your project is ready to use the ZenonSDK!

## Initializing the SDK

Android

Before using the Zenon SDK, a third party application needs to initialize the ZenonPhoneManager. The MainActivity.java contains this initialization process:

mPM.start(this);

iOS

The third party application needs to initialize the Zenon SDK. The Calling Application Delegate APIs should broadcast 4 events from respective positions. The four events to be dispatched from the AppDelegate.m class of calling application are below:

- (void)applicationDidEnterBackground:(UIApplication \*)application {

    [EventsManager dispatchEvent:@"applicationDidEnterBackground" object:self userInfo:nil];

}

- (void)applicationWillEnterForeground:(UIApplication \*)application {

        [EventsManager dispatchEvent:@"applicationWillEnterForeground" object:self userInfo:nil];

}

- (void)applicationDidBecomeActive:(UIApplication \*)application {

    [EventsManager dispatchEvent:@"applicationDidBecomeActive" object:self userInfo:nil];

}

- (void)applicationWillTerminate:(UIApplication \*)application {

    [EventsManager dispatchEvent:@"applicationWillTerminate" object:self userInfo:nil];

}

These events will be received by the ZenonSDK library for configuration purposes.

For further clarity, check the AppDelegate.m class in the bundled ZenonSDKExample directory.

## Setting the Central Login Server

Android

The third party application can set the following parameters in order to customize over default settings of the SDK:

* **Address of the login server** using strings.xml in the res/values folder. The attribute name is ‘*default\_cls’*. Currently, it is set to [*https://cloudroom.saypage.com*](https://cloudroom.saypage.com)
* **Partner name** of the service using strings.xml in the res/values folder. The attribute name is ‘*text\_partner’*. Currently, it is set to [**cloudroom***.*](https://cloudroom.saypage.com)
* **Service name** of the Zenon Service using strings.xml in res/values folder. The attribute name is ‘*servicename*’.Currently, it is set to **saypage.**

After this, the application can proceed with Login and call-setup procedures as outlined in the sample application code.

iOS

The third party application can set the following parameters using standard configuration process through a property or header file (Example application has these settings in AppDelegate.h file) in order to customize over default settings of the SDK:

* **Address of the Load Management Server (LMS).** The attribute name is *‘LmsKey’*. Currently, it is set to [*https://cloudroom.saypage.com*](https://cloudroom.saypage.com)*.*
* **Address of the login server**. The attribute name is ‘*ClsShowUser’*. Currently, it is set to [*https://cloudroom.saypage.com*](https://cloudroom.saypage.com)
* **Partner name** of the service. The attribute name is ‘*Partner’*. Currently, it is set to [**cloudroom***.*](https://cloudroom.saypage.com)
* **Service name** of the Zenon Service. The attribute name is ‘*Service*’.Currently, it is set to **saypage.**

After this, the application can proceed with Login and call-setup procedures as outlined in the sample application code. Code snippet for these settings from the Example is:

[ConfigManager setUserParamForKey:ZS\_CLS\_KEY\_SHOW\_USER value:@CLS\_DEFAULT\_VALUE];

[ConfigManager setUserParamForKey:ZS\_LMS\_KEY value:@CLS\_DEFAULT\_VALUE];

[ConfigManager setUserParamForKey:ZS\_PARTNER\_KEY value:@PARTNER\_DEFAULT\_VALUE];

[ConfigManager setUserParamForKey:ZS\_SERVICE\_KEY value:@SERVICE\_DEFAULT\_VALUE];

## Login to Zenon Server

Once the *default\_cls* is set in the calling application, the calling application can login to the Zenon server as shown in the **MainActivity.java/LoginViewController.m** in the sample application code that ships with the SDK. This login can be typically triggered along with the initialization procedure of the calling application or from a point when the calling application wants to start using zenon SDK based calling services. In order to login to the zenon server, the app needs to call the following API from SDK:

Android

ConnectionManager.loginToServer(ConfigManager.getInstance().getCommonParam(SettingItemNames.settings\_cls).toString(), mCallerText.getText().toString(), mCallerPassText.getText().toString(), MainActivity.this);

This code snippet is taken from MainActivity.java in the bundled example source (src/com/zenon/example).

iOS

 [ConnectionManager login:loginField.text password:passwordField.text optionalParams:nil];

|  |  |
| --- | --- |
| |  | | --- | |  | |

This code snippet is taken from LoginViewController.m in the bundled example source (ZenonSDKExample/ZenonSDKExample).

## Make Video/Voice Calls

Android

Once the user is logged into the Zenon Server, the calling application can provide triggers to initiate Video and Voice calls to our server by using the SDK API. The code snippet showing the call initiation is as below:

String callLineNumber = "1";

mCM.placeCall(((v == mCallVideoButton) ? UserConstants.VIDEO\_CALL : UserConstants.VOICE\_CALL), (mCalleeText.getText().toString()+"#"+callLineNumber), "", "",true, pendingIntent);

Intent callIntent = new Intent(this, CallUIManager.class);

callIntent.putExtra("calledLineNumber", callLineNumber);

this.startActivity(callIntent);

There are two main steps for the initiation:

1. Call placeCall API with appropriate params into ZenonSDK’s CallManager. This initiates the call setup procedures in the SDK.
2. Create and manage the Call GUI by starting the sample call screen activity shipped with this SDK. The calling application can choose to replace the CallUIManager with its own custom CallUI. However, CallUIManager provides a full implementation of call UI so it can be reused by Calling Application as it is.

iOS

Initialize the Call GUI screen:

- (void) getPhoneCallControllerInstance

{ phnCtrl = [PhoneCallViewController sharedInstance];

}

Launch the Call GUI Screen shipped with the SDK:

[phnCtrl initOutboundCall:callType];

Place Call through the Zenon SDK library:

callType = ZENON\_CALL\_TYPE\_VIDEO; //for making a Video Call, or

callType = ZENON\_CALL\_TYPE\_VOICE;//for making a Voice Call

[CallManager placeCall:callType contacts:numberDispTF.text chat\_session\_id:@"" call\_id:@""];

## Receive Incoming Video/Voice Calls

In order to receive and handle an incoming call, the calling application needs to register listeners for IncomingCall events.

Android

EventManager.addListener(this, this.mACTION\_INCOMING\_ResultReceiver, EventDispatcher.ACTION\_INCOMING\_RESULT);

iOS

[CallManager addListener:@selector(handleNotification:) target:self forEvent:@"INCOMING\_CALL"];  
[CallManager addListener:@selector(handleNotification:) target:self forEvent:@"BG\_INCOMING\_CALL"];

Since IOS differentiates between foreground and background modes of application, the calling application needs to register for two different events corresponding to each operating mode of the application as shown above.

Example implementations of event handlers have been provided in respective SDK example source codes for reference.

## Accept/Reject Incoming Video/Voice Calls

Android

lineNumber = "1";//System.currentTimeMillis()+"";

CallManager.getInstance().acceptIncomingCall(lineNumber, null);

CallManager acceptIncomingCall() API can be used to accept an incoming call. Linenumber can be set to “1” by default.

CallManager.getInstance().rejectCall();

CallManager rejectCall() API can be used to reject an incoming call.

iOS

ZENON\_CALL\_TYPE callType = currentCallType;  
[phnCtrl initOutboundCall:callType];  
[CallManager acceptCall:callType lineNumber:@"1"];

CallManager acceptCall API can be used to accept an incoming call. Linenumber can be set to “1” by default.

[CallManager rejectCall];

CallManager rejectCall API can be used to reject an incoming call.

## Switch from Speaker to Headphone and back

In order to listen to audio via a headphone when user plugs in the headphones to the device, there is some routine that needs to be implemented for android by the calling application.

Android

The calling application needs to listen to system event ACTION\_HEADSET\_PLUG.

mHeadphoneReceiver = new HeadphoneReceiver();

registerReceiver(mHeadphoneReceiver, new IntentFilter(Intent.ACTION\_HEADSET\_PLUG));

Sample implementation of the Headphone Receiver handler is provided below. Please refer to Sample application CallUIManager.java for further details.

**private** **class** HeadphoneReceiver **extends** BroadcastReceiver {

**private** **boolean** mStoredSpeakerphoneOnState = **false**;

@Override

**public** **void** onReceive(Context context, Intent intent) {

**int** state = intent.getIntExtra("state",0); // 0 unplugged, 1 plugged, 2 also plugged

**if** (state == 1 || state == 2) { // headphones are plugged in

// force the speakerphone off but store the current state first so we can revert neatly when unplugged

mStoredSpeakerphoneOnState = mSpeakerPhone;

setSpeakerPhone(**false**);

isHeadSetPluggedIn = **true**;

}

**else** { // headphones are not plugged in

// revert back to the stored state

setSpeakerPhone(mStoredSpeakerphoneOnState);

isHeadSetPluggedIn = **false**;

}

}

}

iOS

Zenon SDK for IOS provides *setAudioRoute* API through call manger to switch to/from speaker/headphone. Please refer to API section for iOS SDK for further details.

## Logout from Zenon SDK

Android

This will log out the user from Zenon Server. Calling application can trigger this as a part of its logout /shutdown routine execution. Code snippet showing this is as follows:

mCM.logout();

ConnectionManager.logout();

mPM.stop();

All these code pieces can be referenced from the example code **MainActivity.java** supplied with the SDK.

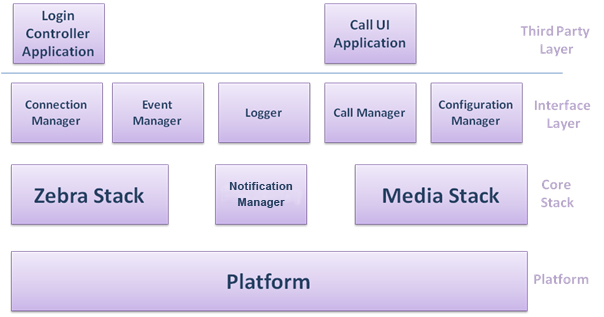
iOS

This will log out the user from Zenon Server. Calling application can trigger this as a part of its logout /shutdown routine execution. Code snippet showing this is as follows:

[ConnectionManager logout];

All these code pieces can be referenced from the example code **HomeViewController.m** supplied with the SDK.

# SDK Architecture



High Level Architecture Diagram of Zenon SDK

The Zenon SDK consists of an Application Interface Layer at the top which exposes the relevant service APIs to third party developers. The main service classes are as below:

1. ConnectionManager: This class provides the service functionality pertaining to user login/logout on Zenon Servers and client session management.
2. EventManager: This class provides service APIs for event mechanism in/out of the SDK. The third party application can subscribe to events from the Zenon SDK to manage UI changes or any other application-specific functionality.
3. Logger: This class provides service APIs for custom logging which can be filtered in or out based on user preferences. This will be helpful in debugging the third party application.
4. CallManager: This is the main Call Control class which provides the third party application to initiate, manage and terminate Voice/Video Call Sessions. The third party app will only need to manage the GUI without bothering with call session setup or media setup.
5. ConfigurationManager: This class provides the third party application a capability to save user-specific data in a persistent storage. This can be used by the application to control login/logout, calls, or any other user-specific functionality. There is a helper class called UserConstants which provides sample declarations of basic application Data. This can be further expanded to include custom data entries that the application may want to use.

In addition to these, there is another set of helper classes which the Zenon SDK provides for third party apps to manage Call UI. A sample implementation of all these classes is shipped with the SDK as CallUIManager. A brief explanation of these helper classes is as below:

1. ZenonLocalVideoView: This class provides the capability to create and customize a local Video window in a Video Call application.
2. ZenonRemoteVideoView: This class provides the capability to create and customize a remote Video window in a Video Call application.
3. ZenonDeviceUtilities: This is a helper class to set the Audio Stream to music or voice call type. Due to certain Android device-specific dependencies, we are setting the stream type to Music for Samsung Galaxy Android devices. For other devices we are using Audio for Voice Call Type.
4. ZenonPhoneCall: This is a helper class to provide an instance of an active call to manage the call session at the application layer.
5. ZenonPhoneManager: This class provides the capability to set the preference order of codec types and initialize the media engine within the SDK.

# Application Requirements

Android

In order to create an application that uses the Zenon SDK for Android, your application has to ensure that it requests certain permissions. This is achieved by adding the following sections to the AndroidManifest.xml:

**Minimum SDK version must be >= 7:**

<uses-sdk android:minSdkVersion=”7”/>

**Required device permissions:**

<uses-permission android:name=”android.permission.INTERNET” />

<uses-permission android:name=”android.permission.RECORD\_AUDIO” />

<uses-permission android:name=”android.permission.MODIFY\_AUDIO\_SETTINGS” />

<uses-permission android:name=”android.permission.PERSISTENT\_ACTIVITY” />

<uses-permission android:name=”android.permission.CAMERA” />

<uses-permission android:name=”android.permission.ACCESS\_NETWORK\_STATE” />

<uses-permission android:name=”android.permission.ACCESS\_CHECKIN\_PROPERTIES” />

<uses-permission android:name=”android.permission.WRITE\_OWNER\_DATA” />

<uses-permission android:name=”android.permission.WRITE\_EXTERNAL\_STORAGE” />

<uses-permission android:name=”android.permission.WAKE\_LOCK” />

<uses-permission android:name=”android.permission.READ\_OWNER\_DATA” />

<uses-permission android:name=”android.permission.SEND\_SMS”/>

<uses-permission android:name=”android.permission.CALL\_PHONE”/>

<uses-permission android:name=”android.permission.VIBRATE” />

<uses-permission android:name=”android.permission.READ\_PHONE\_STATE” />

<uses-permission android:name=”android.permission.AUTHENTICATE\_ACCOUNTS” />

<uses-permission android:name=”android.permission.INTERNET” />

<uses-permission android:name=”android.permission.ACCESS\_NETWORK\_STATE” />

<uses-permission android:name=”android.permission.READ\_CONTACTS” />

<uses-permission android:name=”android.permission.WRITE\_CONTACTS” />

<uses-permission android:name=”android.permission.GET\_ACCOUNTS” />

<uses-permission android:name=”android.permission.MANAGE\_ACCOUNTS” />

<uses-permission android:name=”android.permission.READ\_SYNC\_SETTINGS” />

<uses-permission android:name=”android.permission.WRITE\_SYNC\_SETTINGS” />

<uses-permission android:name="android.permission.GET\_TASKS" />

<uses-permission android:name=”android.permission.ACCESS\_WIFI\_STATE”/>

<uses-permission android:name=”android.permission.UPDATE\_DEVICE\_STATS”/>

<uses-permission android:name=”android.permission.CHANGE\_WIFI\_STATE”/>

<uses-permission android:name=”android.permission.WAKE\_LOCK”/>

<uses-permission android:name=”android.permission.READ\_PHONE\_STATE” />

<uses-permission android:name=”android.permission.AUTHENTICATE\_ACCOUNTS” />

<uses-permission android:name=”android.permission.INTERNET” />

<uses-permission android:name=”android.permission.ACCESS\_NETWORK\_STATE” />

<uses-permission android:name=”android.permission.READ\_CONTACTS” />

<uses-permission android:name=”android.permission.WRITE\_CONTACTS” />

<uses-permission android:name=”android.permission.GET\_ACCOUNTS” />

<uses-permission android:name=”android.permission.MANAGE\_ACCOUNTS” />

<uses-permission android:name=”android.permission.READ\_SYNC\_SETTINGS” />

<uses-permission android:name=”android.permission.WRITE\_SYNC\_SETTINGS” />

<uses-permission android:name="android.permission.GET\_TASKS" />

<uses-permission android:name=”android.permission.ACCESS\_WIFI\_STATE”/>

<uses-permission android:name=”android.permission.UPDATE\_DEVICE\_STATS”/>

<uses-permission android:name=”android.permission.CHANGE\_WIFI\_STATE”/>

<uses-permission android:name=”android.permission.WAKE\_LOCK”/>

**To use the supplied MainActivity and CallUIManager for login/logout & making and receiving calls:**

<activity android:name=”.MainActivity” android:screenOrientation=”portrait” android:label=”@string/app\_name”>

<intent-filter>

<action android:name=”android.intent.action.MAIN” />

<category android:name=”android.intent.category.LAUNCHER” />

</intent-filter>

</activity>

<activity android:name=”.CallUIManager” android:screenOrientation=”portrait” android:launchMode=”singleInstance” android:label=”@string/app\_name” />

iOS

A third party application must specify the below properties in the **info.plist** file:

**Required device capabilities** - armv7

**Required background modes** - App provides Voice over IP services

**Localization native development region** – en.

**Application requires iPhone environment** - YES

**Main nib file base name** – MainWindow

**Application does not run in background** – NO

# Supported Devices

Android

The Zenon SDK is compatible with all Android versions from Eclair 2.1 to KitKat 4.4 platform versions of Android.

The SDK has been tested on:

* Samsung Galaxy S
* Samsung Galaxy S2
* Samsung Galaxy S3
* HTC EVO
* Samsung Galaxy Tab
* Samsung Galaxy Tab 10.1
* Motorola XOOM
* Samsung Galaxy Note
* Samsung Galaxy Note 2

And many more android devices

iOS

The Zenon SDK is compatible with all iOS versions up to 7.0.

The SDK has been tested on:

* iPhone 4
* iPhone5
* iPad
* iPad 2
* iPad3

# SDK API Reference

## Android

### ConfigManager Class

The ConfigManager class allows storage and retrieval of parameters or data, which are identified by simple strings such as their names. If supported by the platform, these parameters can be stored in the persistent storage.

#### Configuration Parameters

These are client parameters, and are independent of the user.

**Examples:** CLS, last-logged in user, etc.

|  |  |
| --- | --- |
| Static Setter | void ConfigManager.setCommonParam(String name, Object value) |
| Static Getter | Object ConfigManager.getCommonParam(String name) |

#### Login-Specific Parameters

These are user-specific parameters that should be cleared as soon as the user logs out.

**Examples:** username, avatar, SIP settings, etc.

|  |  |
| --- | --- |
| Static Setter | void ConfigManager.setUserParam(String name, Object value) |
| Static Getter | Object ConfigManager.getUserParam(String name) |
| Static Initializer | public static void ConfigManager.initSettings(Context context) |
| Static First Install Checker | public static boolean ConfigManager.isFirstInstall(Context context) |
| Static Default Setter | public void ConfigManager.createDefaultSettings() throws InvalidSettingsException |

### EventDispatcher Class

The SDK should provide a mechanism to share information within the client by letting any class dispatch events that other components in the client can listen to. The events that are dispatched are identified using simple string names, and each of them carries an optional data object that is passed as a parameter to the listener method.

The EventDispatcher class is the base class of all such classes that dispatch events and let other components set a listener to these events. Event names can be reused in these classes, but should be unique within a class.

The EventsManager class provides the following protected methods to dispatch, subscribe to and unsubscribe from events. A subclass should provide its own public APIs to let other components take advantage of this mechanism.

|  |  |
| --- | --- |
| Dispatcher | **protected** **void** \_dispatchEvent(String eventName, Object data) |
| Subscriber | **protected** **void** \_addListener(String eventName, listenerMethod) |
| Unsubscriber | **protected** **void** \_removeListener(String eventName, listenerMethod) |

#### Listener Methods

A listener method should have the following signature, so that the optional event data can be passed as a parameter:

|  |
| --- |
| **void** listenerMethod(Object eventData) {  //body  } |

#### Challenges

The following challenges need to be considered while designing the EventsManager class. In the worst case, depending on the platform, events might have to be replaced with callback methods.

* Effect of the sequences of the listener subscriptions, and whether the sequence should be controlled or not.
* Memory leakage - references to the subscribed listeners may prevent the listening object from memory de-allocation. Weak references to be used wherever possible.
* Performance and memory constraints.

### EventsManager Class

The EventsManager is a singleton class that extends the [EventDispatcher](#_EventDispatcher_Class_1) class to provide static methods that allow any component in the client to share internal events.

#### Dispatching Internal Events

|  |  |
| --- | --- |
| Public API | void EventManager.dispatchEvent(String eventName, Object data) |

#### Subscribing to Internal Events

|  |  |
| --- | --- |
| Public API | void EventManager.addListener(String eventName, listener) |

#### Unsubscribing from Internal Events

|  |  |
| --- | --- |
| Public API | void EventManager.removeListener(String eventName, listener) |

### Zebra Class

Zebra, an acronym of Zenon Event Broker Request Architecture, is a lightweight XML format to share information between Zenon Servers and Clients, only after an authentication is successful (refer to section [Signing In](#_12.2_Signing_In) for more details on authentication and login). It follows the following format:

|  |
| --- |
| <zebra username=*""* service=*""* event=*""* anchor=*""* error\_code=*""*>  <!--OPTIONAL-->  <zconfig>  <!--Zero or more-->  <key mode=*"update|insert|delete"*>value</key>  </zconfig>    <!--OPTIONAL-->  <zevent>  <!--Zero or more-->  <key mode=*"update|insert|delete"*>value</key>  </zevent>    </zebra> |

Note that the nodes in this format have a maximum depth of 2. The root element <zebra> has two optional elements:

1. <zconfig> To pass configuration parameters.
2. <zevent> To pass events and data.

Each of the above elements can contain a single level of “child” elements, and every node in these elements can again have a “mode” attribute, which decides the way the parameter is read.

Zebra objects are sent and received by the [ConnectionManager](#_ConnectionManager_Class_1) singleton.

#### Creating a Zebra Object

Zebra objects can be constructed using either of the two constructors:

|  |
| --- |
| **new** Zebra(String event) |

Where the argument event provides the value of the “event” attribute in the Zebra. The “username” and “service” attributes of the Zebra are obtained during the login process.

Further information can be set in the Zebra object as described in the next section.

#### Setting Information in a Zebra Object

All the following setter methods return the Zebra object itself, to allow the chaining of multiple setters in the same statement.

|  |  |
| --- | --- |
| Setter Method | Description |
| public Zebra(String username\_, String service\_, String event\_) | Constructor API for creating a new Zebra Object |
| public static synchronized Zebra ParseZEBRAMessage(String zebraMessage) | API for Parsing an XML Zebra String into a Zebra object |
| public void setFormatType(String mFmt) | API for setting Format Type (fmt=””) of a Zebra message |
| public void setEventType(String mEvent) | API for setting Event Type (event=””) of a Zebra message |
| public void setZEventMode(String mode) | API for setting Event Mode (mode=””) of a Zebra message |
| public static void setZEventKeyValue(String keyName, String value) | Adds/updates a <zevent> key with no “mode” attribute. This can internally call the setZEvent() method by skipping the optional mode argument. |
| public static void setZConfigKeyValue(String keyName, String value) | Adds/updates a <zconfig> key with no “mode” attribute. This can internally call the setZEvent() method by skipping the optional mode argument. |

#### Retrieving Information from a Zebra Object

|  |  |
| --- | --- |
| Getter Method | Description |
| public String getFormatType() | Gets the Zebra Format Type (1.0 or 2.0) |
| public String getEventType() | Gets the Zebra Event Type |
| public String getZEventMode() | Gets the Zebra Mode |
| public static String getZEventKeyValue(String keyName) | Returns a <zevent> key's value. |
| public String toXML() | Returns the string representation of the Zebra |
| **public** **static** String getZConfigKeyValue(String keyName) | Returns a <zconfig> key's value. |

### ConnectionManager Class

The ConnectionManager is a singleton class that manages all connections to and from the servers, and provides static API methods to perform these functions. It extends the [EventDispatcher](#_EventDispatcher_Class_1) class to broadcast server-related events that any component in the client can subscribe to or unsubscribe from. The dispatcher method is not a public API in this class.

#### Server Events

The ConnectionManager class dispatches the following events:

| Event Name | Event Data | Description |
| --- | --- | --- |
| LOGIN\_SUCCESS | Array of username and nickname | Dispatched when the user has successfully logged in. |
| LOGIN\_FAILED | Array of username and reason for failure | Dispatched when the login process failed. |
| CONNECTION\_LOST | null | Dispatched when the connection to the Zenon Server is lost. |
| CONNECTION\_MADE | null | Dispatched when the connection to the Zenon Server has been established. |
| LOGOUT\_SUCCESS | null | Dispatched when the user has successfully logged out. |
| LOGOUT\_FAILED | Reason for failure | Dispatched when the logout process has failed. |
| ZEBRA:[Zebra Name] | Zebra object | Dispatched when a Zebra with “event” [Zebra Name] is received from the server. |

Components can subscribe to these events using the following static method:

|  |  |
| --- | --- |
| Public API | public static void addListener(Context context,BroadcastReceiver receiver,String event) |

To unsubscribe themselves, components can call the following static method:

|  |  |
| --- | --- |
| Public API | public static void removeListener(Context context,BroadcastReceiver receiver) |

#### Signing In

The ConnectionManager class will have the following static method to let a user login to the CLS:

|  |
| --- |
| public static void loginToServer(String server, String username, String password,Context context) |

#### Signing Out

The ConnectionManager class will have the following static method to let the user logout of the CLS:

|  |
| --- |
| public static ZenonStateHandler getZenonStateHandler()  ConnectionManager.logout() |

This method should be non-blocking, and the result can be asynchronously dispatched as events (LOGOUT\_SUCCESS, LOGOUT\_FAILED).

#### Receiving Asynchronous Notifications

The client will remain connected to one of the available Zenon Servers in order to receive messages, such as incoming calls, new IM, friend requests, and more.

The connection will be a low-power HTTP long poll on Android.

The ConnectionManager starts this connection as soon as the user successfully signs in, and stops it when the user signs out. The ConnectionManager also dispatches appropriate events (CONNECTION\_LOST, CONNECTION\_MADE) regarding the connection to the Zenon Server.

#### Send Zebra

The ConnectionManager class will have the following static method to send a [Zebra](#_Zebra_Class_1) to the CLS:

|  |
| --- |
| public static String[] sendZebra(Zebra zebra)  public static String[] sendZebratoCLS(Zebra zebra) |

#### Receive Zebra

The ConnectionManager receives [Zebras](#_Zebra_Class_1) from the server, either in the response of receiving asynchronous notifications, or in response to a Zebra sent, as described in the previous two sections. Once it receives a Zebra, it dispatches an event with the name as per that in the Zebra's “event” attribute, prefixed with “ZEBRA:”, and the data being the Zebra object itself. Classes can thus subscribe to listen to a Zebra, say with “event” attribute as “ZEBRA\_EVENT\_NAME”, as follows:

|  |
| --- |
| ConnectionManager.addListener("ZEBRA:ZEBRA\_EVENT\_NAME", listenerMethod);  **void** listenerMethod(Object data) {  //example code:  var Zebra zebra = (Zebra) data;  var String zebraEventName = zebra.getZebraAttribute("event"); //=> "ZEBRA\_EVENT\_NAME"  } |

### CallManager Class

The CallManager is a singleton class that manages all voice and video calls, and provides static API methods to do these functions. It extends the [EventDispatcher](#_EventDispatcher_Class_1) class to broadcast call-related events that any component in the client can subscribe to or unsubscribe from. The dispatcher method is not a public API in this class.

#### Call Events

The CallManager class dispatches the following events:

| Event Name | Event Data | Description |
| --- | --- | --- |
| Constants.INCOMING\_CALL | Incoming Call Zebra | Dispatched when an incoming call is received. |
| Constants.OUTGOING\_CALL | ZenonPhoneCall | Dispatched when an outgoing call is made. |
| Constants.ACCEPT\_CALL | Nil | Dispatched when a call is accepted. |
| Constants.END\_CALL | Nil | Dispatched when a call is ended by the user or by the other party. |
| Constants.REJECT\_CALL | Nil | Dispatched when a call is rejected. |
| Constants.CALL\_ESTABLISHED | Nil | Dispatched when a call is established. |

Components can subscribe to these events using the following static method:

|  |  |
| --- | --- |
| Public API | **void** CallManager.addListener(String eventName, listener) |

To unsubscribe themselves, components can call the following static method:

|  |  |
| --- | --- |
| Public API | **void** CallManager.removeListener(String eventName, listener) |

#### Initiating/Managing/Terminating a Voice/Video call

|  |  |
| --- | --- |
| Public API | **public** **void** placeCall(String call\_type, String destination,String session\_id, String call\_Id, PendingIntent pendingIntent)  Parameters:  call\_type: UserConstants.*VIDEO\_CALL,*UserConstants.*VOICE\_CALL*  destination: This is the list of contactid and lineNumbers in a comma separated format(e.g. a@b.com#1, [c@d.com#2](mailto:c@d.com#2) or [event\_109291094\_13599@vidiconf.com](mailto:event_109291094_13599@vidiconf.com))  session\_id: A unique identifier at the app layer to identify the newly created call session (e.g “”)  call\_Id: A unique identifier for the call. This will not change for until call exists. Calling application can use “” if it doesn’t need to identify call.  pendingIntent: This is a pending Intent required to handle the Incoming Call messages in the Media Plane |
| **Public API** | **public** **void** endCall() |
| **Public API** | **public** **void** acceptIncomingCall(String lineNumber) |
| **Public API** | **public** **void** rejectCall() |
| **Public API** | **public** **void** getCall() |
| **Public API** | **public** **void** takeCall(Intent intent) |
| **Public API** | **public static [CallManager](file:///E:\\apps\\Latest_Code_V2\\ZMobile\\Apps\\ZSip\\Android\\v3\\ZenonSDK\\doc\\com\\zenon\\sdk\\core\\CallManager.html" \o "class in com.zenon.sdk.core)**getInstance() |
| **Public API** | **public static ZenonPhoneCall.CameraOrientation** getRemoteOrientation() |
| **Public API** | **public void** setMuteMic(**final boolean** muted)  muted: true/false – true for muting the microphone, false for un-muting the microphone. |
| **Public API** | **public void** setMuteCam(**final boolean** toMute)  toMute: true/false – true for muting the Camera, false for un-muting the Camera. |

**Note:** For a list of Call End Reason Codes, please see [Appendix A](#_Appendix_A:_Call).

### Logger Class

The Logger class allows organized and controlled logging, with detailed information on the location and level/type of the log.

A good way to display a log line would be:

<log type> | <message> | <filename> | <line number> | <method>

An example log line:

|  |
| --- |
| FATAL | This is a fatal log message | path/filename | 291 | testMethod |

The separators will allow easy imports in Excel and similar applications; and displaying the call location, which may vary from platform to platform, will help in debugging.

The following table describes the static APIs, log levels and types, and their descriptions.

|  |  |  |  |
| --- | --- | --- | --- |
| Level | Type | API | Description |
| 0 | FATAL | Logger.fatal(String log) | Severe errors that cause premature termination, and cannot be ignored. |
| 1 | ERROR | Logger.error(String log) | Runtime errors or unexpected conditions that might need some inspection. |
| 2 | WARN | Logger.warn(String log) | Runtime situations that are undesirable or unexpected, but not necessarily "wrong". |
| 3 | INFO | Logger.info(String log) | Interesting runtime events. |
| 4 | DEBUG | Logger.debug(String log) | Detailed information on the flow through the system. |
| 5 | TRACE | Logger.trace(String log) | More detailed information, especially for developers. |

There are two other static APIs available to the calling application to enable/disable logging in ZenonSDK. They are as below:

void Logger.modifyLogging(boolean isLogging);

boolean Logger.isLoggingEnabled();

Logger configurations, like log levels etc, can be set/retrieved as configuration parameters in the ConfigManager class.

## iOS

### ConfigManager Class

The ConfigManager class allows storage and retrieval of parameters or data, which are identified by simple strings such as their names. If supported by the platform, these parameters can be stored in the persistent storage.

#### Configuration Parameters

These are client parameters, and are independent of the user.

**Examples:** CLS, last-logged in user, etc.

|  |  |
| --- | --- |
| Static Setter | +(void) setCommonParamForKey:(NSString \*)key value:(NSString \*)val; |
| Static Getter | +(NSString \*) getCommonParamForKey:(NSString \*)key defaultValue:(NSString\*)defaultVal;  //the object "defaultValue" is returned if no param with "paramName" is available. |

#### Login-Specific Parameters

These are user-specific parameters that should be cleared as soon as the user logs out.

**Examples:** username, avatar, SIP settings, etc.

|  |  |
| --- | --- |
| Static Setter | +(void) setUserParamForKey:(NSString \*)key value:(NSString \*)val; |
| Static Getter | +(NSString \*) getUserParamForKey:(NSString \*)key defaultValue:(NSString \*)defaultVal;  //the object "defaultValue" is returned if no param with "paramName" is available. |

### EventDispatcher Class

The SDK should provide a mechanism to share information within the client by letting any class dispatch events that other components in the client can listen to. The events that are dispatched are identified using simple string names, and each of them carries an optional data object that is passed as a parameter to the listener method.

The EventDispatcher class is the base class of all such classes that dispatch events and let other components set a listener to these events. Event names can be reused in these classes, but should be unique within a class.

The EventsManager class provides the following protected methods to dispatch, subscribe to and unsubscribe from events. A subclass should provide its own public APIs to let other components take advantage of this mechanism.

|  |  |
| --- | --- |
| Dispatcher | **protected** **void** \_dispatchEvent(String eventName, Object data) |
| Subscriber | **protected** **void** \_addListener(String eventName, listenerMethod) |
| Unsubscriber | **protected** **void** \_removeListener(String eventName, listenerMethod) |

#### Listener Methods

A listener method should have the following signature, so that the optional event data can be passed as a parameter:

|  |
| --- |
| **void** listenerMethod(Object eventData) {  //body  } |

#### Challenges

The following challenges need to be considered while designing the EventsManager class. In the worst case, depending on the platform, events might have to be replaced with callback methods.

* Effect of the sequences of the listener subscriptions, and whether the sequence should be controlled or not.
* Memory leakage - references to the subscribed listeners may prevent the listening object from memory de-allocation. Weak references to be used wherever possible.
* Performance and memory constraints.

### EventsManager Class

The EventsManager is a singleton class that extends the [EventDispatcher](#_EventDispatcher_class) class to provide static methods that allow any component in the client to share internal events.

#### Dispatching Internal Events

|  |  |
| --- | --- |
| Public API | + (void)dispatchEvent:(NSString\*)event userInfo:(id)data; |

#### Subscribing to Internal Events

|  |  |
| --- | --- |
| Public API | + (void)addListener:(SEL)selector target:(id)object forEvent:(NSString\*)event; |

#### Unsubscribing from Internal Events

|  |  |
| --- | --- |
| Public API | + (void)removeListener:(id)target forEvent:(NSString\*)event; |

### Zebra Class

Zebra, an acronym of Zenon Event Broker Request Architecture, is a lightweight XML format to share information between Zenon Servers and Clients, only after an authentication is successful (refer to section [Signing In](#_Signing_In) for more details on authentication and login). It follows the following format:

|  |
| --- |
| <zebra username=*""* service=*""* event=*""* anchor=*""* error\_code=*""*>  <!--OPTIONAL-->  <zconfig>  <!--Zero or more-->  <key mode=*"update|insert|delete"*>value</key>  </zconfig>    <!--OPTIONAL-->  <zevent>  <!--Zero or more-->  <key mode=*"update|insert|delete"*>value</key>  </zevent>    </zebra> |

Note that the nodes in this format have a maximum depth of 2. The root element <zebra> has two optional elements:

1. <zconfig> To pass configuration parameters.
2. <zevent> To pass events and data.

Each of the above elements can contain a single level of “child” elements, and every node in these elements can again have a “mode” attribute, which decides the way the parameter is read.

Zebra objects are sent and received by the [ConnectionManager](#_ConnectionManager_class) singleton.

#### Creating a Zebra Object

Zebra objects can be constructed using either of the two constructors:

|  |
| --- |
| **new** Zebra(String event) |

Where the argument event provides the value of the “event” attribute in the Zebra. The “username” and “service” attributes of the Zebra are obtained during the login process.

Further information can be set in the Zebra object as described in the next section.

#### Setting Information in a Zebra Object

All the following setter methods return the Zebra object itself, to allow the chaining of multiple setters in the same statement.

|  |  |
| --- | --- |
| Setter Method | Description |
| - (void) setZebraAttribute:(NSString \*)name value:(NSString \*)value; | Adds/updates a <zebra> attribute. |
| - (void) setZConfigKey:(NSString \*)key value:(NSString \*)value; | Adds/updates a <zconfig> key with no “mode” attribute. This can internally call the setZConfig() method by skipping the optional mode argument. |
| - (void) setZConfigKey:(NSString \*)key value:(NSString \*)value mode:(NSString \*)mode; | Adds/updates a <zconfig> key. The “mode” attribute is optional. |
| - (void) setZEventKey:(NSString \*)key value:(NSString \*)value; | Adds/updates a <zevent> key with no “mode” attribute. This can internally call the setZEvent() method by skipping the optional mode argument. |
| - (void) setZEventKey:(NSString \*)key value:(NSString \*)value mode:(NSString \*)mode; | Adds/updates a <zevent> key. The “mode” attribute is optional. |

#### Retrieving Information from a Zebra Object

|  |  |
| --- | --- |
| Getter Method | Description |
| - (NSString \*) getZebraAttribute:(NSString \*)name; | Returns a <zebra> attribute value. |
| - (NSString \*) getZEventKeyValue:(NSString \*)key; | Returns a <zevent> key's value. |
| - (NSString \*) getZeventValue:(NSString \*)key; | Returns a <zevent> value. |
| - (NSString \*) getZeventMode:(NSString \*)key; | Returns a <zevent> key's “mode” attribute value. |
| - (NSString \*) getZconfigMode:(NSString \*)key; | Returns a <zconfig> key's “mode” attribute value. |
| -(NSString \*) getZconfigValue:(NSString \*)key; | Returns a <zconfig> key’s value. |
| - (NSString \*) toXML; | Returns the string representation of the Zebra. |

### ConnectionManager Class

The ConnectionManager is a singleton class that manages all connections to and from the servers, and provides static API methods to perform these functions. It extends the [EventDispatcher](#_EventDispatcher_class) class to broadcast server-related events that any component in the client can subscribe to or unsubscribe from. The dispatcher method is not a public API in this class.

#### Server Events

The ConnectionManager class dispatches the following events:

| Event Name | Event Data | Description |
| --- | --- | --- |
| LOGIN\_SUCCESS | Array of username, nickname | Dispatched when the user has successfully logged in. |
| LOGIN\_FAILED | Array of username and reason for failure | Dispatched when the login process failed. |
| CONNECTION\_LOST | null | Dispatched when the connection to the Zenon Server is lost. |
| CONNECTION\_MADE | null | Dispatched when the connection to the Zenon Server has been established. |
| LOGOUT\_SUCCESS | null | Dispatched when the user has successfully logged out. |
| LOGOUT\_FAILED | Reason for failure | Dispatched when the logout process has failed. |
| ZEBRA:[Zebra Name] | Zebra object | Dispatched when a Zebra with “event” [Zebra Name] is received from the server. |

Components can subscribe to these events using the following static method:

|  |  |
| --- | --- |
| Public API | + (void)addListener:(SEL)selector target:(id)object forEvent:(NSString\*)event; |

To unsubscribe themselves, components can call the following static method:

|  |  |
| --- | --- |
| Public API | + (void)removeListener:(id)target forEvent:(NSString\*)event; |

#### Signing In

The ConnectionManager class will have the following static method to let a user login to the CLS:

|  |
| --- |
| - (void) login; |

#### Signing Out

The ConnectionManager class will have the following static method to let the user logout of the CLS:

|  |
| --- |
| - (void) logout; |

This method should be non-blocking, and the result can be asynchronously dispatched as events (LOGOUT\_SUCCESS, LOGOUT\_FAILED).

#### Receiving Asynchronous Notifications

The client will remain connected to one of the available Zenon Servers in order to receive messages, such as incoming calls, new IM, friend requests, and more.

The connection will be a background VOIP socket on iOS.

The ConnectionManager starts this connection as soon as the user successfully signs in, and stops it when the user signs out. The ConnectionManager also dispatches appropriate events (CONNECTION\_LOST, CONNECTION\_MADE) regarding the connection to the Zenon Server.

#### Send Zebra

The ConnectionManager class will have the following static method to send a [Zebra](#_Zebra_class) to the CLS:

|  |
| --- |
| + (void)sendZebra:(Zebra\*)zebra; |

#### Receive Zebra

The ConnectionManager receives [Zebras](#_Zebra_class) from the server, either in the response of receiving asynchronous notifications, or in response to a Zebra sent, as described in the previous sections. Once it receives a Zebra, it dispatches an event with the name as per that in the Zebra's “event” attribute, prefixed with “ZEBRA:”, and the data being the Zebra object itself. Classes can thus subscribe to listen to a Zebra, say with “event” attribute as “ZEBRA\_EVENT\_NAME”, as follows:

|  |
| --- |
| ConnectionManager.addListener("ZEBRA:ZEBRA\_EVENT\_NAME", listenerMethod);  **void** listenerMethod(Object data) {  //example code:  var Zebra zebra = (Zebra) data;  var String zebraEventName = zebra.getZebraAttribute("event"); //=> "ZEBRA\_EVENT\_NAME"  } |

### CallManager Class

The CallManager is a singleton class that manages all voice and video calls, and provides static API methods to do these functions. It extends the [EventDispatcher](#_EventDispatcher_class) class to broadcast call related events that any component in the client can subscribe to or unsubscribe from. The dispatcher method is not a public API in this class.

#### Call Events

The CallManager class dispatches the following events:

| Event Name | Event Data | Description |
| --- | --- | --- |
| INCOMING\_CALL | NSDictionary with keys and values as follows:  key - "contactAddress" - (NSString \*)caller address  key - "contactType" - (NSString\*) caller contact type  key - "callType" - (NSNumber \*) call\_type (enum ZENON\_CALL\_TYPE - defined in CallManager.h)  key - "chatSessionId" - (NSString\*) chat session id  key - "avatarId" - (NSString\*) Avatar id of caller | Dispatched when an incoming call is received. |
| OUTGOING\_CALL | Nil | Dispatched when an outgoing call is made. |
| CALL\_ENDED | NSDictionary with keys and values as follows:  key - "reason" - (NSNumber \*) call end reason (enum ZENON\_CALL\_END\_REASON - defined in CallManager.h) | Dispatched when a call is ended by the user or by the other party. |
| CALL\_QUALITY\_UPDATE\_LOW | Nil | Call quality gone below threshold |
| CALL\_QUALITY\_UPDATE\_NORMAL | Nil | Call quality gone above normal |
| CALL\_METADATA\_CHANGED | NSDictionary with keys and values as follows:  key - "remoteRotation" - (NSNumber \*) remote rotation (enum ZENON\_ROTATION\_TYPE - defined in CallManager.h)  key - "remoteCameraType" - (NSNumber \*) remote camera type (enum ZENON\_CAMERA\_TYPE - defined in CallManager.h) | Remote party's camera or orientation changed |

Components can subscribe to these events using the following static method:

|  |  |
| --- | --- |
| Public API | + (void)addListener:(SEL)selector target:(id)object forEvent:(NSString\*)event; |

To unsubscribe themselves, components can call the following static method:

|  |  |
| --- | --- |
| Public API | + (void)removeListener:(id)target forEvent:(NSString\*)event; |

#### 

#### Initiating/Managing/Terminating a Voice/Video call

| Public API | Description |
| --- | --- |
| + (void)addListener:(SEL)selector target:(id)object forEvent:(NSString\*)event; |  |
| + (void)removeListener:(id)target forEvent:(NSString\*)event; |  |
| + (void) placeCall:(ZENON\_CALL\_TYPE)type contacts:(NSString \*)contacts chat\_session\_id:(NSString \*)session\_id call\_id:(NSString \*)cid; | Type: ZENON\_CALL\_TYPE\_VOICE/ZENON\_CALL\_TYPE\_VIDEO  contacts: A string containing comma-separated contacts lists to place calls to. Each contact in the list can have an application-defined line number (an integer) in the end with a # separator. If no line number is specified, line number 1 is assumed.  chat\_session\_id: The session ID of the one-to-one or group chat session.  call\_id: An application specified for the application to identify the call. |
| + (void) endCall; |  |
| + (void) acceptCall:(ZENON\_CALL\_TYPE)callType lineNumber:(NSString \*)line; | Type: ZENON\_CALL\_TYPE\_VOICE / ZENON\_CALL\_TYPE\_VIDEO  lineNumber: The line number for the incoming call to be accepted |
| + (void) rejectCall; |  |
| + (void) muteMic:(BOOL)mute; | mute: YES/NO |
| + (void) setAudioRoute:(ZENON\_AUDIO\_ROUTE)route; | route: ZENON\_AUDIO\_ROUTE\_SPEAKER\_PHONE / ZENON\_AUDIO\_ROUTE\_HANDSET |
| + (void) showLocalVideo:(AVCaptureVideoPreviewLayer \*)layer; | layer: The AVCapture layer used by application to render local video |
| + (void) showRemoteVideo:(CALayer \*)layer; | layer: The layer used by application to render remote video |

**Note:** For a list of Call End Reason Codes, please see [Appendix A](#_Appendix_A:_Call).

### Logger Class

The Logger class allows organized and controlled logging, with detailed information on the location and level/type of the log.

A good way to display a log line would be:

<log type> | <message> | <filename> | <line number> | <method>

An example log line:

|  |
| --- |
| FATAL | This is a fatal log message | path/filename | 291 | testMethod |

The separators will allow easy imports in Excel and similar applications; and displaying the call location, which may vary from platform to platform, will help in debugging.

The following table describes the static APIs, log levels and types, and their descriptions.

|  |  |  |  |
| --- | --- | --- | --- |
| Level | Type | API | Description |
| 0 | FATAL | + (void) fatal☹NSString \*)message; | Severe errors that cause premature termination, and cannot be ignored. |
| 1 | ERROR | + (void) error☹NSString \*)message; | Runtime errors or unexpected conditions that might need some inspection. |
| 2 | WARN | + (void) warn☹NSString \*)message; | Runtime situations that are undesirable or unexpected, but not necessarily “wrong”. |
| 3 | DEBUG | + (void) debug☹NSString \*)message; | Detailed information on the flow through the system. |
| 4 | INFO | + (void) info☹NSString \*)message; | Interesting runtime events. |

Logger configurations, like log levels etc, can be set/retrieved as configuration parameters in the ConfigManager class.

# 

# Appendix A: Call End Reason Codes

Call end reason codes used in Disconnect Zebra.

|  |  |
| --- | --- |
| Code | Explanation |
| 0 | Local endpoint application cleared call |
| -1 | Local endpoint did not accept call OnIncomingCall()=PFalse |
| -2 | Local endpoint declined to answer call |
| -3 | Remote endpoint application cleared call |
| -4 | Remote endpoint refused call |
| -5 | Remote endpoint did not answer in required time |
| -6 | Remote endpoint stopped calling |
| -7 | Transport error cleared call |
| -8 | Transport connection failed to establish call |
| -9 | Gatekeeper has cleared call |
| -10 | Call failed as could not find user (in GK) |
| -11 | Call failed as could not get enough bandwidth |
| -12 | Could not find common capabilities |
| -13 | Call was forwarded using FACILITY message |
| -14 | Call failed a security check and was ended |
| -15 | Local endpoint busy |
| -16 | Local endpoint congested |
| -17 | Remote endpoint busy |
| -18 | Remote endpoint congested |
| -19 | Could not reach the remote party |
| -20 | The remote party is not running an endpoint |
| -21 | The remote party host is offline |
| -22 | The remote failed temporarily, app may retry |
| -23 | The remote ended the call with unmapped Q.931 cause code |
| -24 | Call cleared due to an enforced duration limit |
| -25 | Call cleared due to invalid conference ID |
| -26 | Call cleared due to missing dial tone |
| -27 | Call cleared due to missing ringback tone |
| -28 | Call cleared because the line is out of service |
| -29 | Call cleared because another call is answered |
| -30 | Call cleared because gatekeeper admission request failed |

# Appendix B: Zebra API

For a list of all Zebras, please click here: <http://www.zenontel.com/user/docs/zebra/>

This common XML format is known as Zebra, which stands for Zenon Event Broker Request Architecture.

Zebra allows the different platform components (across Client and Server) to publish events to one another and pass data within those events. It is similar to SOAP but simplified and lightweight. It also encapsulates basic dataset operations: that is, the insert, update and delete commands which you may well know from SQL.

On one hand, Zebra is nothing more than a convention for defining the XML used throughout the platform, but through its generic nature it becomes extremely powerful. All the XML messages follow the same consistent pattern so it's therefore possible to create a generic parser which can parse any Zebra XML structure, without specific knowledge of the message, and create a Zebra object which contains all the data.

Here you have an example which shows the basic anatomy of a Zebra message:

|  |
| --- |
| *<?xml version="1.0" encoding="UTF-8"?> <zebra fmt='1.0' username='sessionId' service='serviceA‘ event='eventX‘ error\_code='0'>     <zevent>         <key1>value1</key>         <key2>value1</key2>     </zevent> </zebra>* |

|  |  |
| --- | --- |
| **Attribute/Node** | **Definition** |
| username | This attribute is a session identifier for a user/device context. |
| service | This attribute is the name of a service, allowing the server to run multiple services in parallel if required. |
| event | This attribute specifies the method to be invoked or event to be fired by the receiver. |
| zevent | This node contains the key/value pairs, which make up the data portion of the Zebra message to be used by the receiver when processing the message. |
| error\_code | This attribute indicates a possible error in performing a Zebra request; zero means no error and a list of defined errors is used across the API. Errors are most likely to occur in response to a request. |
| anchor | This attribute is effectively a timestamp that will only return information from the server that has been modified since the anchor. |

**Device Optimizations**

|  |  |  |
| --- | --- | --- |
| * Query Limit * Max Size * Anchors |  |  |

The Gateway offers the following features to ensure that devices can control the amounts of data they receive and that they do not receive data they already have.

**Query Limit:** Allows the clients to request the maximum number of results to be returned in a result set, and offsets into that result set so that it may page through large amounts of information without needing to store it all locally. This can give you a Google Maps effect on your EPG where only specific chunks of information surrounding your current position are retrieved.

**Max Size:** Allows a client to dictate the maximum size a Zebra should be. So when large update lists need to be sent to the client, they will be broken down into a number of Zebras to ensure they are all below the limit.

**Anchors:** When requesting list data such as contacts and call logs, a client can supply an anchor which is effectively a timestamp. The server will only send data which has been modified since the anchor. This avoids a client continually needing to download its address book (for example) each time the user logs in, as it is able to store such information locally for offline use.

**Data Models**

This message shows a typical Zebra event that a client might receive when the user's list of favourites has changed, possibly on another device. The list\_mode here says 'insert'. Insert tells the client to insert these rows into its favourites data model, and in turn fire an event to the views that might be interested to know if the list of favourites has changed. The ID attribute is the primary key for each row. If the list\_mode is 'delete' then these IDs should be removed from the model. If the list\_mode is 'update' then the entire list in the client should be replaced with this list.

|  |
| --- |
| *<zebra fmt="2.0" username="MySessionId” event="FAVOURITE\_UPDATE" error\_code="0” anchor=“2011-10-01 12:00:30”> <zevent list\_mode=”insert”>     <obj id=”1” cid=”1” favoritePosition =”1”/>     <obj id=”23” cid=”23” favoritePosition =”2”/>     [...] </zevent> </zebra>* |

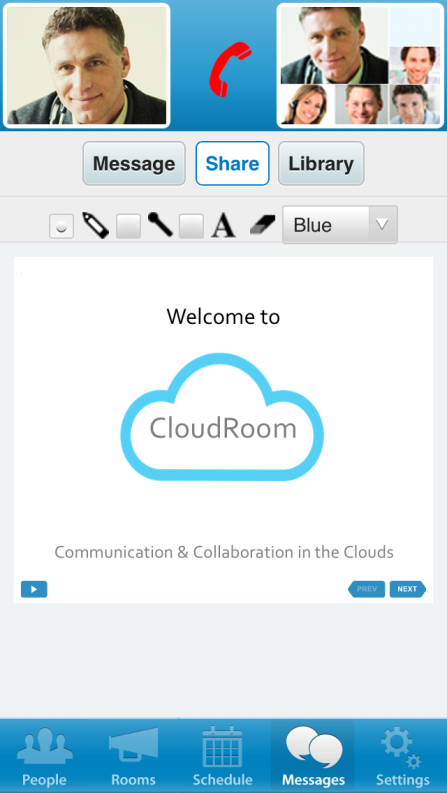
This Zebra shows a client querying a data model in the server. This uses the SQL query syntax 'select', 'where' and 'order by'. Below, a client is calling the getChannels API to retrieve channel information. As you can see, clients can request the data they require. To process this request, the iiTVG would query its internal copy of the EPG, a data set which is shared across all users, and then it would apply an in-memory second query to inject all the personalised data into the result set. Personalised data includes Favourites, Bookmarks, PCR settings, PVR information and search history.

|  |
| --- |
| *<?xml version="1.0" encoding="UTF-8"?> <zebra fmt="2.0" username="MySessionId" event="com.nsn.zebra.iiTV#getChannels"> <zevent>     <select>cname, favorite, favoritePosition, cnumber</select>     <where>subscribed='true' and crating>=4</where>          <orderby>favorite, favoritePosition, cnumber</orderby>          <limit>10,5</limit> </zevent> </zebra>* |

# Appendix C: Configurable Components (Available upon request)

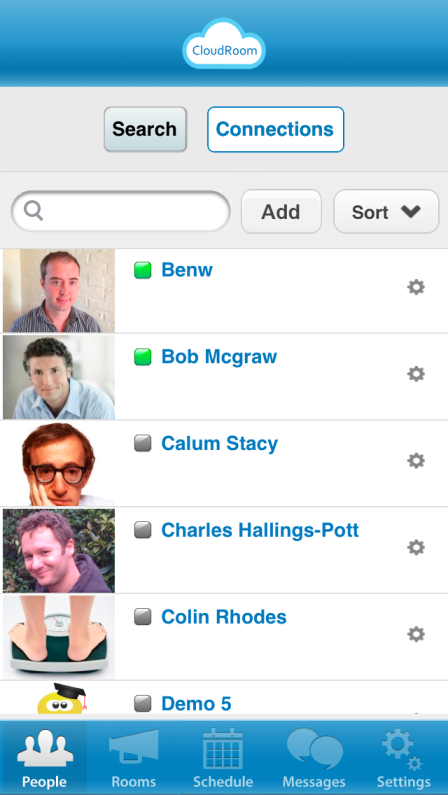
Whiteboard

Whiteboard is an application feature where people can share images, texts, screens through a whiteboard during a live video/voice call. The image below shows it in action:



Presence, Contacts and Friend List

This is an application that allows us to create a friend list through invitations, get/share presence updates and IM friends. The image below shows it in action:



Full Source of Saypage Application built on the Zenon SDK

Our flagship Saypage Application has been built using the Zenon SDK core. It’s full source code release is available upon request.

